Lab4 - K-way Graph Partitioning Using JaBeJa

ID2222 Data Mining

Yage Hao, Hongyi Luo

07 December 2020

1 Introduction

In this assignment, we follow [F. Rahimian, et al., JA-BE-JA: A Distributed Algorithm for Balanced Graph Partitioning, SASO2013] and [F Rahimian et al., A distributed algorithm for large-scale graph partitioning (Links to an external site.), ACM Transactions on Autonomous and Adaptive Systems (TAAS) 10 (2), 12] implements the JA-BE-JA algorithm.

The task can be divided into 3 parts, the first part is the basic implementation of the algorithm. In the second part, we try to change the parameters and introduce the annealing algorithm. In the third part, we define our own acceptance probability function to achieve more good performance.

2 How to Run the Code

To run the code follow the steps above:

Requirements:JDK 8 , Ubuntu system or any other Linux core system

You may first need to install the dependencies below:

```
sudo apt install default-jre
sudo apt install default-jdk
sudo apt install maven
sudo apt install gnuplot
sudo apt install xdg-utils
```

If you want to change the mode depends on task like “task1”,”task2”, you can access to the CLI.Java to switch to other mod

```
@Option(name = "-task", usage = "Number of tasks.")
private static String TASK = "task1";
```

The possible mods are currently "task1",”task2”,”task2r” whcih means task2 with restart and “task3” which aim to bonus task.

Same, if you want to change any parameters related to the execution, if you also need to modify these classes below in CLI.Java.

After that, you can run the following command to test the code, but beware, when you first time run them, chmod +X “file.sh” might be needed.

```
>> ./compile.sh
>> ./run -graph ./graphs/3elt.graph
>>./plot .sh output/result
```

* the result means the file name you generate.

3 Algorithm Explanation

Algorithm 1 JA-BE-JA Algorithm.

Require: Any node p in the graph has the following methods:

- $getNeighbors()$: returns p's neighbors.
- $getSample()$: returns a uniform sample of all the nodes.
- $getDegree(c)$: returns the number of p's neighbors that have color c.

```
1: //Sample and Swap algorithm at node p
2: procedure SAMPLEANDSWAP
3:     partner ← FindPartner(p.getNeighbors(), T_r)
4:     if partner = null then
5:         partner ← FindPartner(p.getSample(), T_r)
6:     end if
7:     if partner ≠ null then
8:         color exchange handshake between p and partner
9:     end if
10:    T_r ← T_r − δ
11:    if T_r < 1 then
12:        T_r ← 1
13:    end if
14: end procedure

15: //Find the best node as swap partner for node p
16: function FINDPARTNER(Node[] nodes, float T_r)
17:    highest ← 0
18:    bestPartner ← null
19:    for q ∈ nodes do
20:        d_pp ← p.getDegree(p.color)
21:        d_qq ← q.getDegree(q.color)
22:        old ← d_pp^α + d_qq^α
23:        d_pq ← p.getDegree(q.color)
24:        d_qp ← q.getDegree(p.color)
25:        new ← d_pq^α + d_qp^α
26:        if (new × T_r > old) ∧ (new > higest) then
27:            bestPartnere ← q
28:            highest ← new
29:        end if
30:    end for
31:    return bestPartner
32: end function
```

Because the algorithm is not very difficult and here we only consider the situation in task1, we can say the main idea is “distributed" "parallel" and “self-improve”. It means that even if we don’t have a whole view of the graph, each node can still improve its own “edge-cut” situation by detecting and testing the neighbor nodes and edge combinations (with color).

For task2, the main change is we introduce the temperature to avoid local-maximum, we can also try restarting with several epochs to test the performance. In the last bonus task, the idea is to change the AP(acceptance probability) in order for the algorithm to get faster coverage.

4 Code Explanation

The code is divided into jebeja.java, jebejaTask2.java and jebejaTask2restart.java.

For the task1 codes, there are **sampleAndSwap(...)** methods and the **findPartner(...) method to be care**

```java
/**
 * Sample and swap algorithm at node p
 * @param nodeId
 */
protected void sampleAndSwap(int nodeId) {
  Node partner = null;
  Node nodep = entireGraph.get(nodeId);

  if (config.getNodeSelectionPolicy() == NodeSelectionPolicy.HYBRID
          || config.getNodeSelectionPolicy() == NodeSelectionPolicy.LOCAL) {
    // swap with random neighbors
    partner = findPartner(nodeId, getNeighbors(nodep));
  }

  if (config.getNodeSelectionPolicy() == NodeSelectionPolicy.HYBRID
          || config.getNodeSelectionPolicy() == NodeSelectionPolicy.RANDOM) {
    // if local policy fails then randomly sample the entire graph
    if(partner == null)
      partner = findPartner(nodeId, getSample(nodeId));
  }

  // swap the colors
  if(partner != null){
    int thisColor = nodep.getColor();
    nodep.setColor(partner.getColor());
    partner.setColor(thisColor);
    numberOfSwaps++;
  }

}
```

```java
public Node findPartner(int nodeId, Integer[] nodes){

  Node nodep = entireGraph.get(nodeId);

  Node bestPartner = null;

  // Task1
  double highestBenefit = 0;


  for (Integer node:nodes) {
    Node nodeq = entireGraph.get(node);

      int dpp = getDegree(nodep, nodep.getColor());
      int dqq = getDegree(nodeq, nodeq.getColor());

      // dpp dqq
      double oldValue = (Math.pow(dpp, config.getAlpha()) + Math.pow(dqq, config.getAlpha()));
      int dpq = getDegree(nodep, nodeq.getColor());
      int dqp = getDegree(nodeq, nodep.getColor());
      // dpq dqp
      double newValue = (Math.pow(dpq, config.getAlpha()) + Math.pow(dqp, config.getAlpha()));
      // Task 1


    if(newValue*this.T > oldValue && newValue > highestBenefit){
      bestPartner = nodeq;
      highestBenefit = newValue;
    }
  }
  return bestPartner;
}
```

Just simply follow the paper instruction.

For task2 and bonus tasks, we introduce the SA tech and AP function.

```java
/**
 * Simulated analealing cooling function
 */
protected void saCoolDown() {

  float tMin = 0.0001f;
  if (this.T > tMin) {
    this.T = T - config.getDelta();
    if (this.T < tMin) System.out.println("\n\n\nReached Tmin\n\n\n");
  }
  if (this.T < tMin)
    this.T = tMin;
}
```

```java
  //-----------------------------------------------------------------
  public JabejaTask2WithRestart(HashMap<Integer, Node> graph, Config config) {
    super(graph, config);
    config.setRounds(10000);
  }


  //-----------------------------------------------------------------
  public void startJabeja() throws IOException {
    for (round = 0; round < config.getRounds(); round++) {
      for (int id : entireGraph.keySet()) {
        sampleAndSwap(id);
      }
      if (roundRestart>0 && round >= firsRestart) {
          if (round % roundRestart == 0) {
              this.T = config.getTemperature();
              this.config.setDelta(this.config.getDelta()/this.config.getTAlpha());
          }
      }
      //one cycle for all nodes have completed.
      //reduce the temperature
      saCoolDown();
      report();
    }
  }
}
```

```java
public Node findPartner(int nodeId, Integer[] nodes){

  Node nodep = entireGraph.get(nodeId);

  Node bestPartner = null;

  // Task1
  double highestBenefit = 0;


  for (Integer node:nodes) {
    Node nodeq = entireGraph.get(node);

    int dpp = getDegree(nodep, nodep.getColor());
    int dqq = getDegree(nodeq, nodeq.getColor());

    // dpp dqq
    double oldValue = (Math.pow(dpp, config.getAlpha()) + Math.pow(dqq, config.getAlpha()));
    int dpq = getDegree(nodep, nodeq.getColor());
    int dqp = getDegree(nodeq, nodep.getColor());
    // dpq dqp
    double newValue = (Math.pow(dpq, config.getAlpha()) + Math.pow(dqp, config.getAlpha()));
    // Task 1



    double acceptProbability = this.calculateAcceptanceProbability(oldValue, newValue);

    if (acceptProbability >= this.random.nextDouble() && newValue > highestBenefit) {
      bestPartner = nodeq;
      highestBenefit = newValue;
    }
  }


  return bestPartner;
}
```

5 Results and Analysis

Task 1:

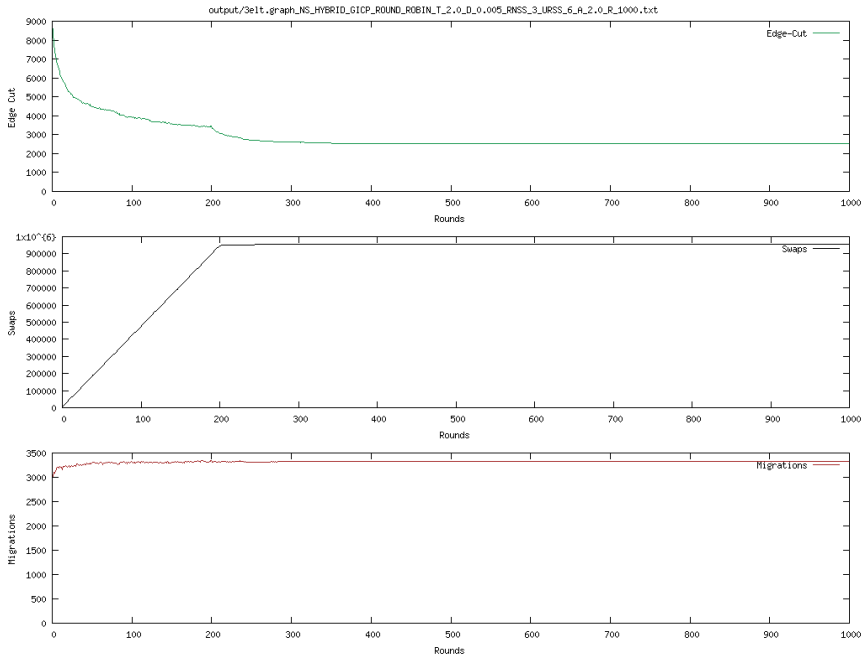
In this part we are supposed to:

- show the test results when we apply 4-way graph partitioning using original ja-be-ja algorithm with parameters suggested by the paper on three different datasets 3elt, add20 and Facebook.

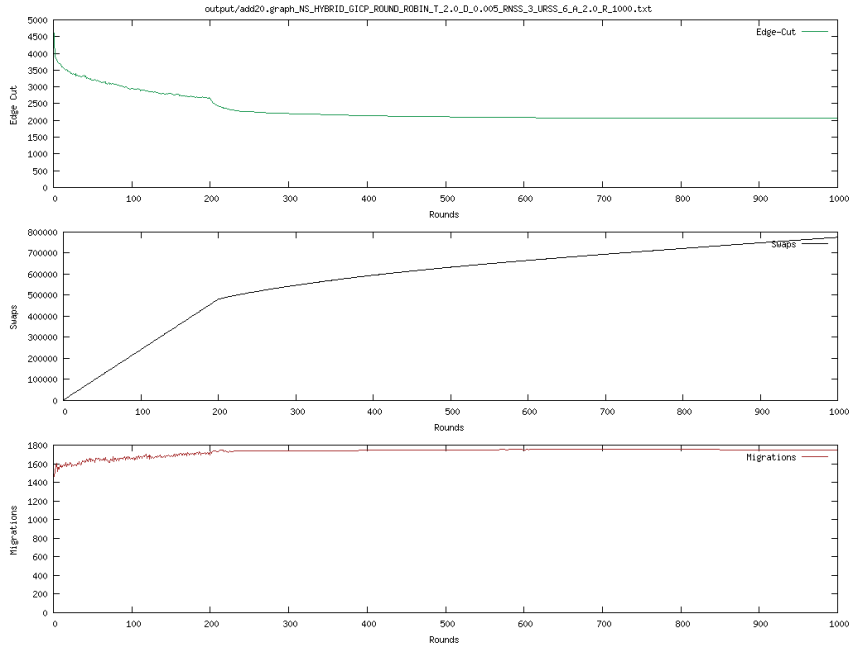
There are some general configurations we hold:

- K = 4, K-way graph partitioning, this parameter should always keep constant.
- Sampling policy = H, shown in the paper that the hybrid heuristic (H) always produces the best performance.
- For original ja-be-ja algorithm (linear SA method): T0 = 2, alpha = 2, delta = 0.03.

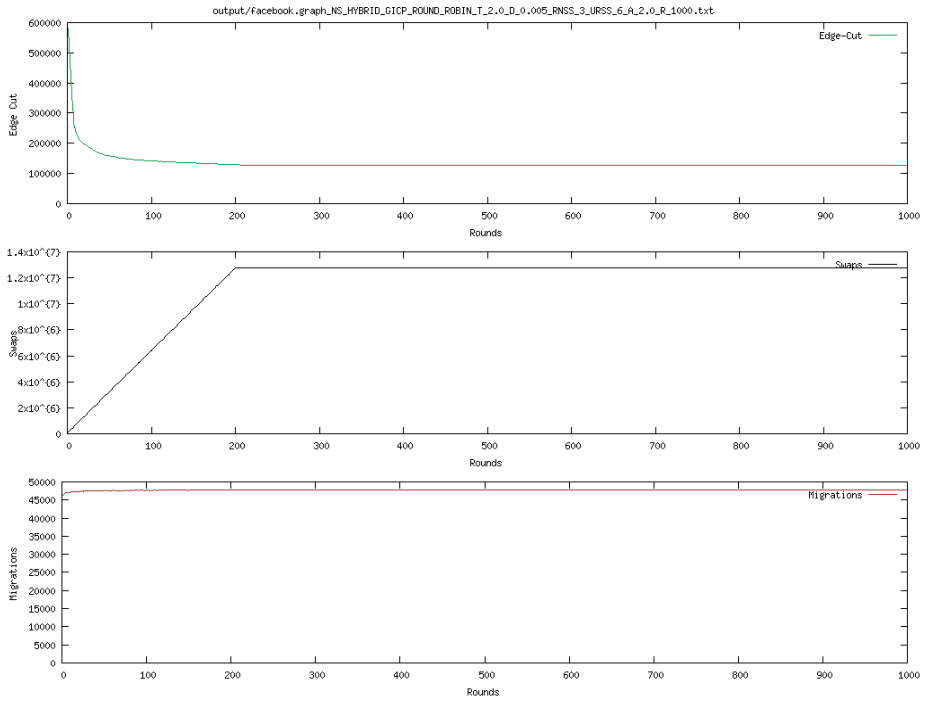
Here are the result plots:



Task1 3elt



Task1 Add



Task1 Facebook

Task 2.1:

In this part we are supposed to:

- analyze how a different simulated mechanism affects the rate of convergence. Remember that when using this method, the maximum initial temperature is 1.
- tweak different parameters.

In general:

- Dataset = 3elt.
- K = 4, K-way graph partitioning, this parameter should always keep constant.
- Sampling policy = H, shown in the paper that the hybrid heuristic (H) always produces the best performance.

Specific for linear SA method:

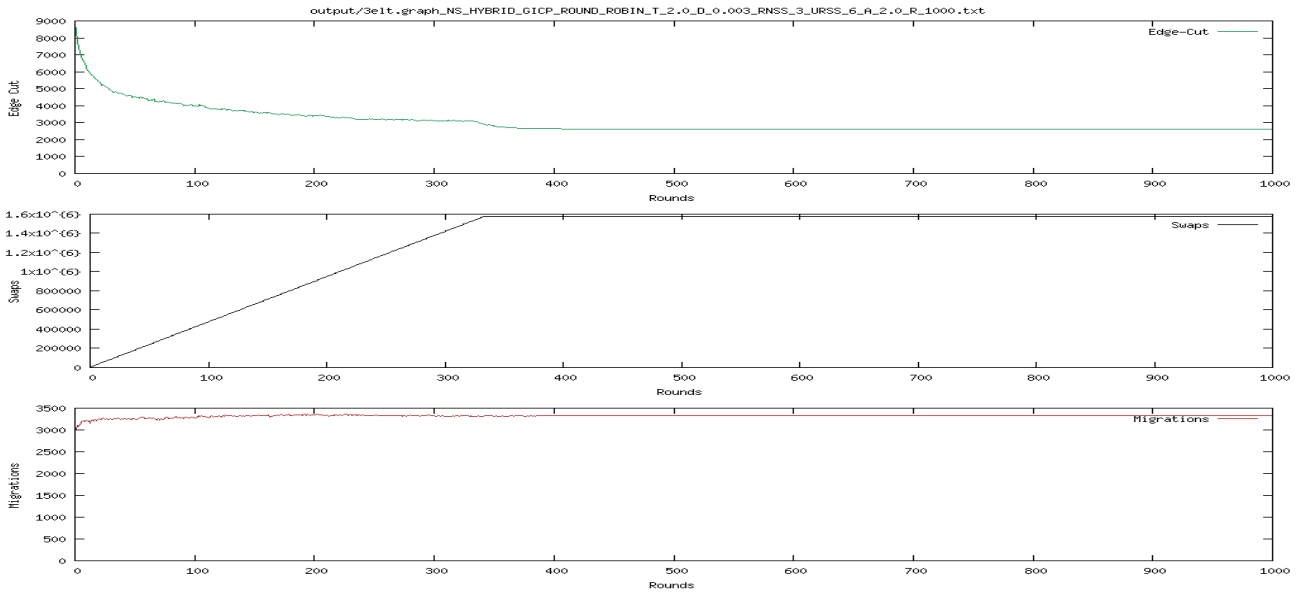
- T0 = 2, starting temperature, keep constant.
- Alpha = 2, a parameter of the swapping condition, keep constant, already shown in paper that linear SA with alpha = 2 will produce the best performance on datasets 3elt.

- Delta = [0.003, 0.9, 0.99], adjustable parameter, the speed of the cooling process.

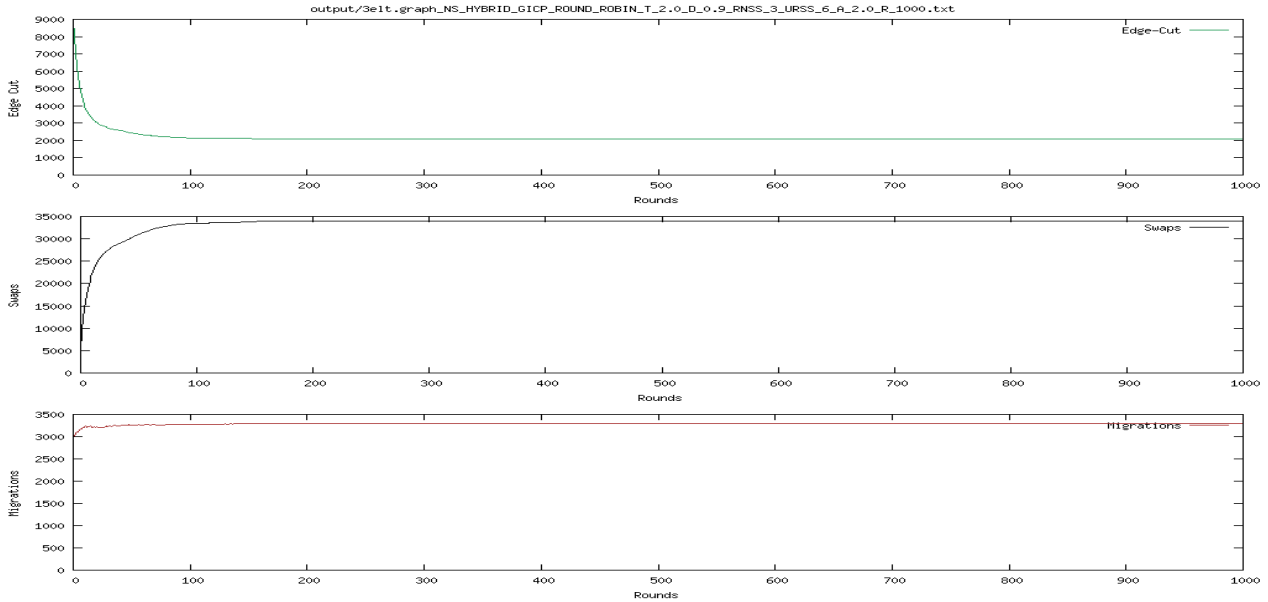
Specific for exponential SA method:

- T0 = 1, starting temperature, keep constant.
- Delta = [0.003, 0.9, 0.99], adjustable parameter, the speed of the cooling process.

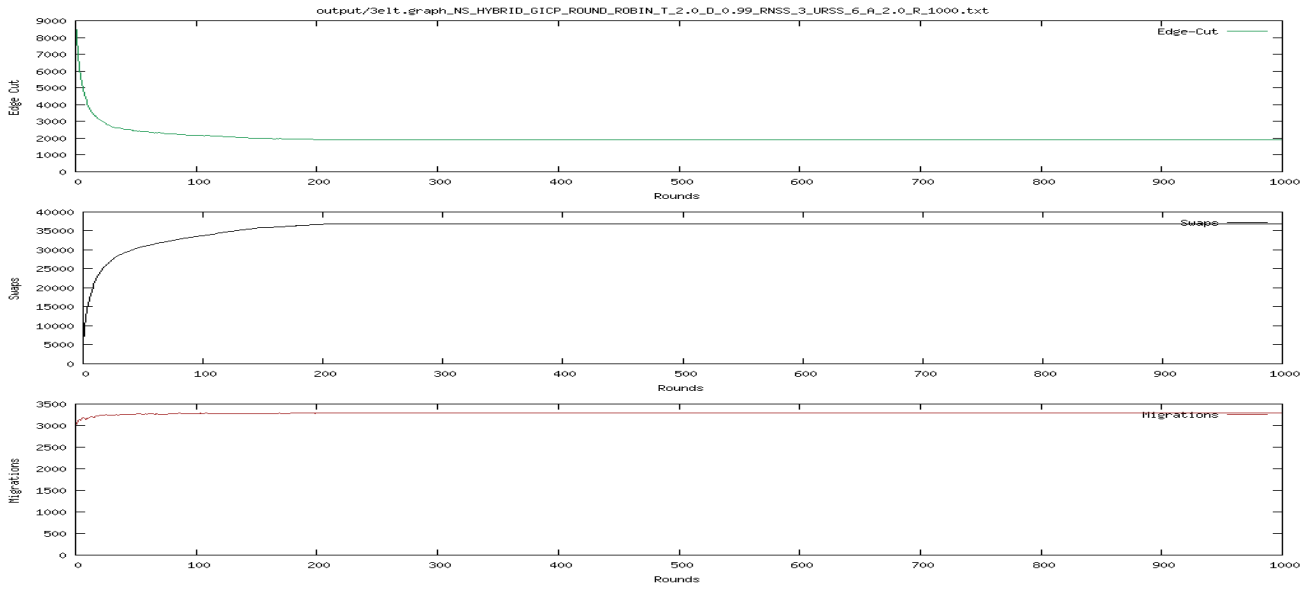
We apply the above parameter tweaking on our dataset. Here are the result plots:



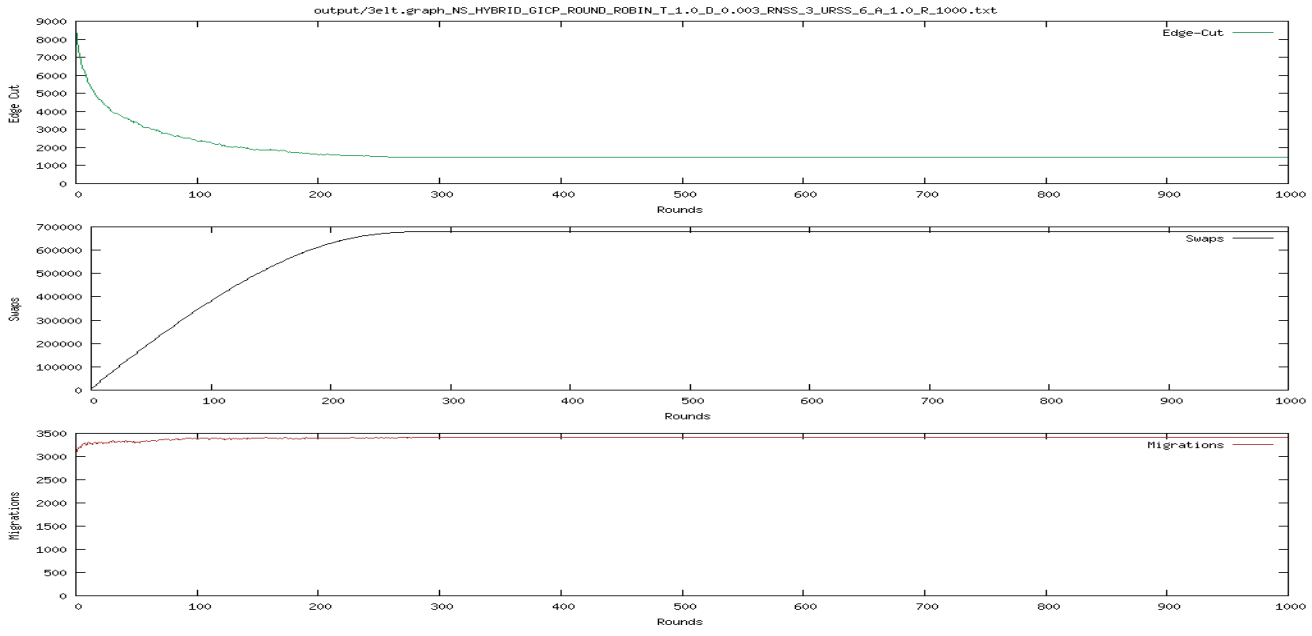
Linear with D= 0.003



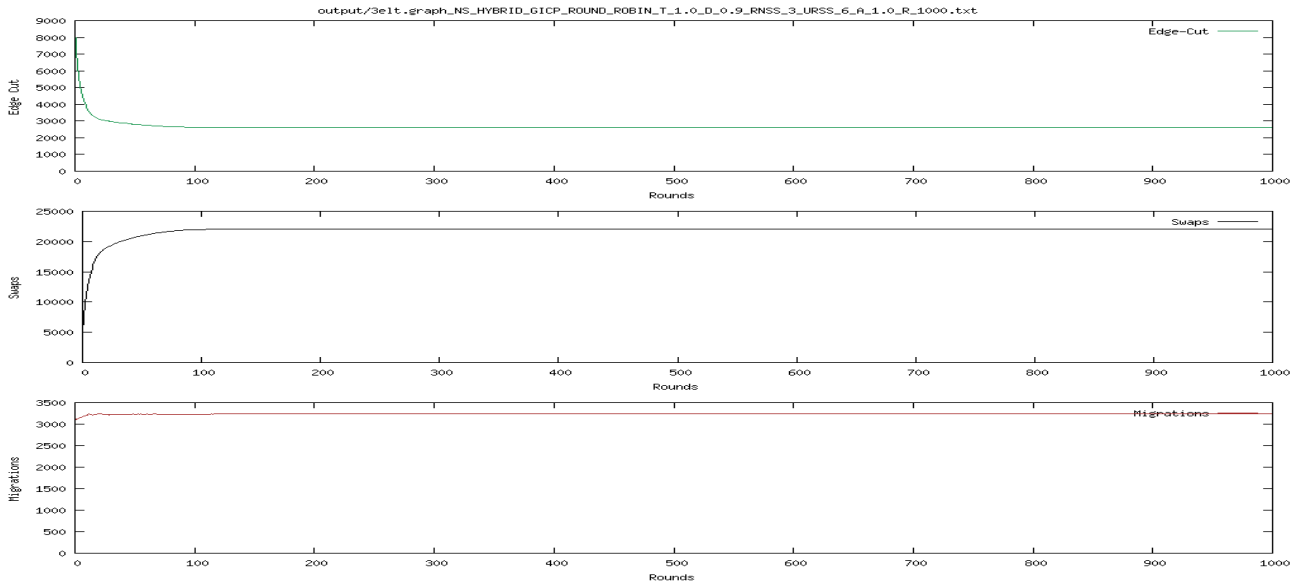
Linear with D= 0.9



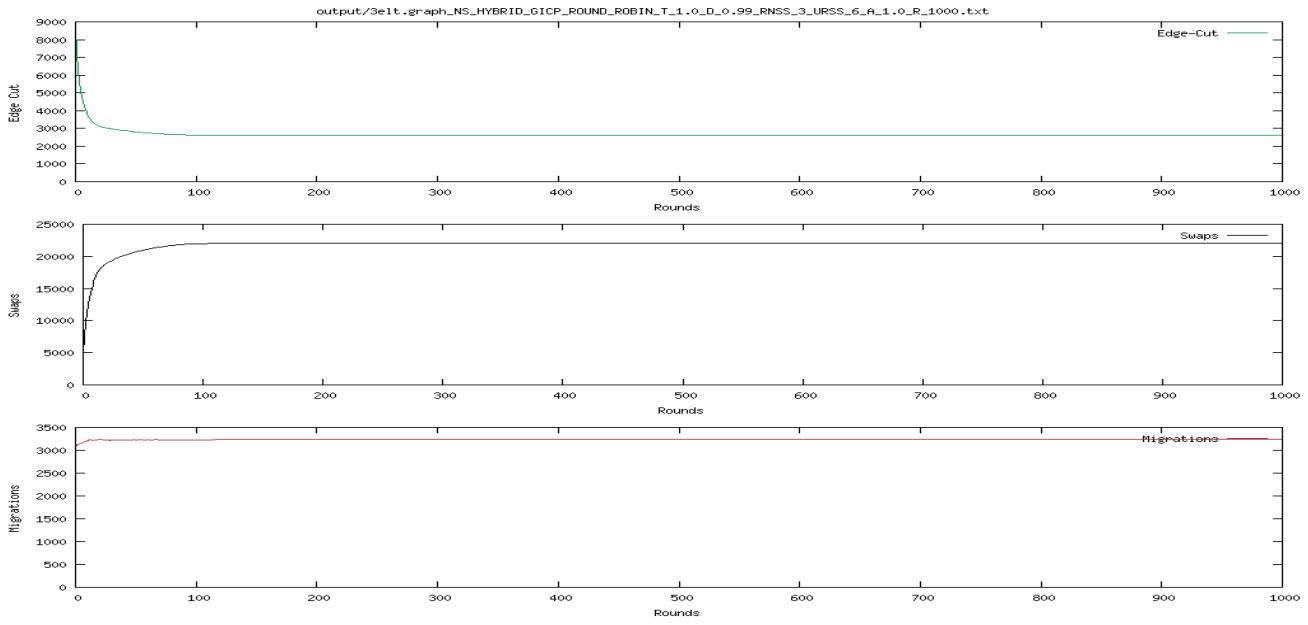
Linear with D= 0.99



exponential with D=0.003



exponential with D=0.9



exponential with D=0.99

In conclusion:

- The algorithm using exponential SA method converges faster than the linear SA one.
- Delta is the key parameter which affects the performance of partitioning algorithm with exponential simulated annealing. In detail, as long as we decrease delta towards the minimum achievable value 0, the improved algorithm spends more time producing results with larger number of swaps

and smaller edge-cut. This also proves that delta represents a trade-off between quality of partitioning and running time.

Task 2.2:

In this part, we are supposed to:

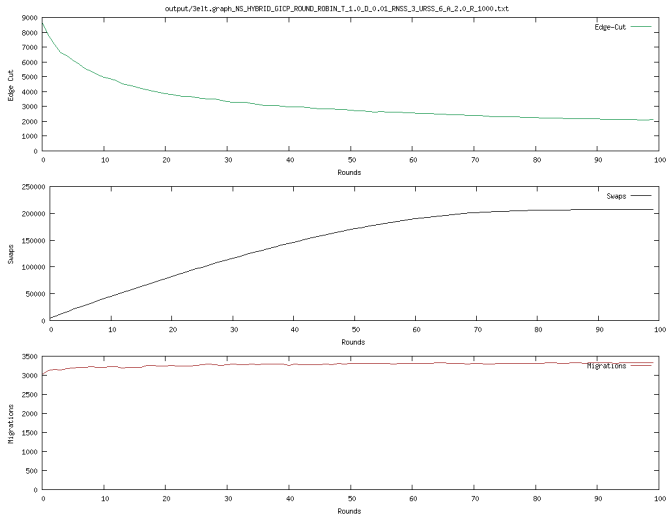
- investigate how the Ja-Be-Ja algorithm behaves when the simulated annealing is restarted after Ja-Be-Ja has converged.
- Experiment with different parameters and configurations to find lower edge cuts.

We set the below configurations:

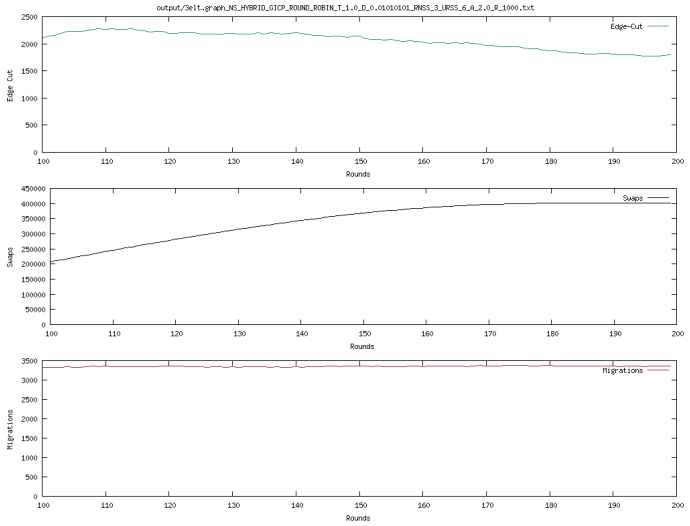
- Dataset = 3elt
- K = 4, K-way graph partitioning
- Sample polity = H, hybrid heuristic
- SA method = exponential simulated annealing method
- T0 = 1, delta = 0.01, then after 100 rounds the temperature will cool down to 0 and no more bad-swaps will be accepted.
- Restart period = [100, 200], adjustable parameter, after each period of rounds we restart the SA process.

We apply the above configurations and adjust parameters on our dataset. Here are the result plots: (NB: we ran for round=1000 in total, here we only present part of the resulting plots.)

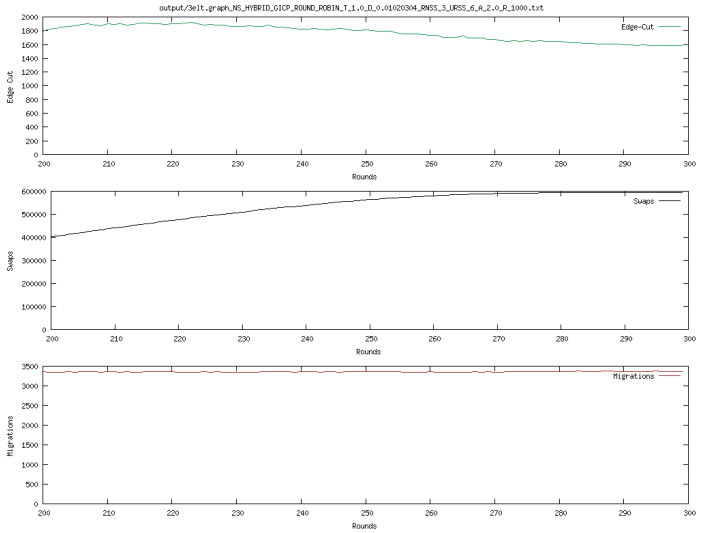
- With restart period = 100:
    - Initial 100 rounds:



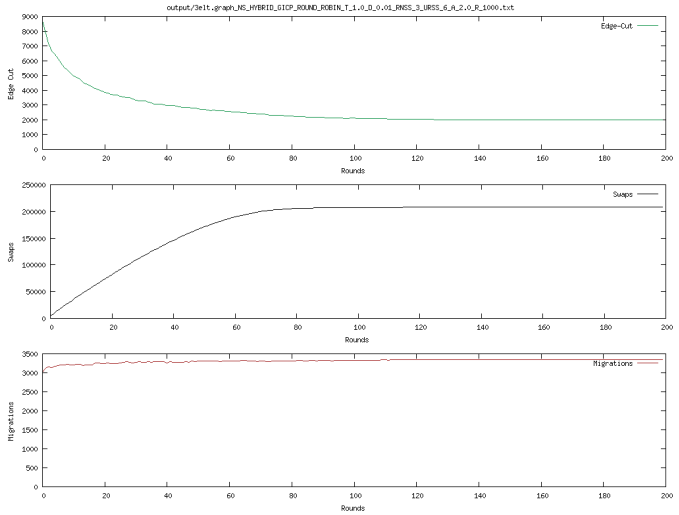
- round 100-200 after the first restart:



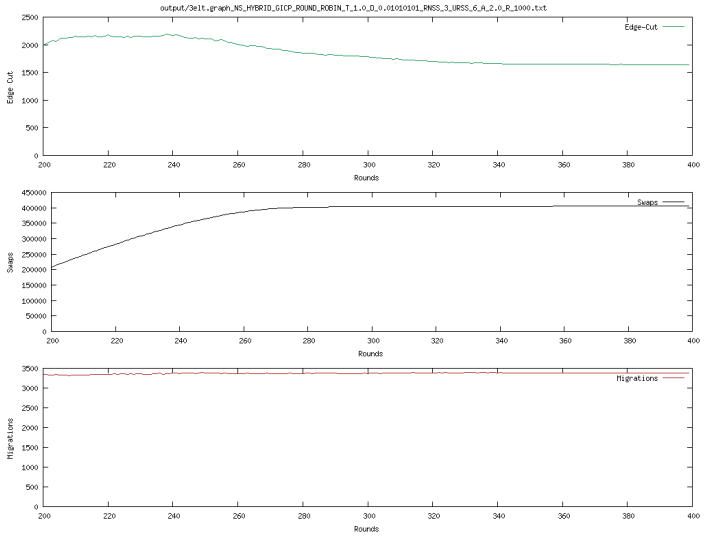
- round 200-300 after the second restart:



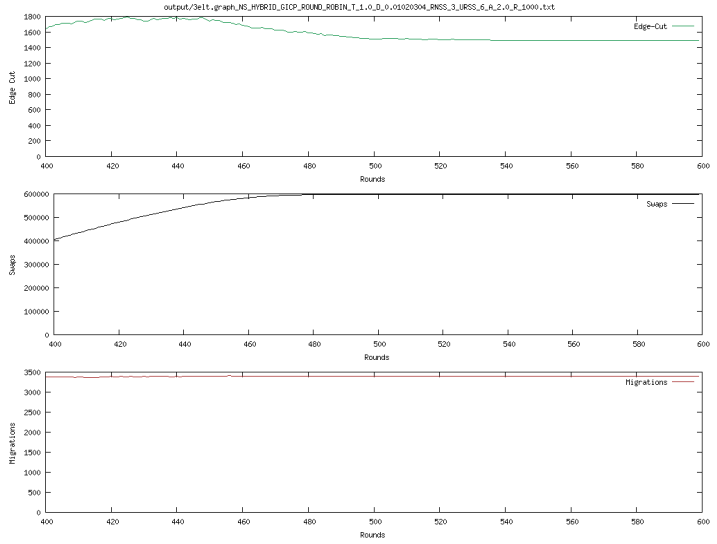
- With restart period = 200:
  - Initial 200 rounds:



- round 200-400 after first restart:



- round 400-600 after second restart:



In conclusion, we find that after each restart, the edge-cut will grow up a little and then decrease to converge. Thus in the long term, the algorithm still provides a decreasing trend in edge-cut. And also since we set T0=1 and delta=0.01, the number of swap maximally is 100. Thus for the first 100 rounds after the restart, the number of swap will always increase, while after that it keeps stable.

Optional Task:

In this part we are supposed to:

- define our own acceptance probability function
- evaluate how this change affects the performance of graph partitioning

We define the new acceptance probability function as:

$$\frac{1}{1+e^{\frac{Eold-Enew}{T}}}$$

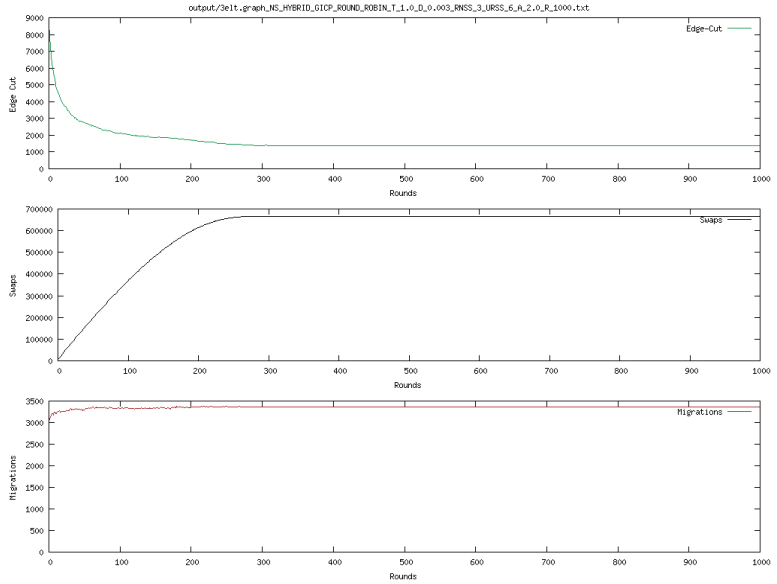
Intuitively, If E_old is larger than E_new, indicating the state is worse, according to the above formula, the acceptance probability will be smaller, and there is unlikely to swap to the worse state. The larger the difference between E_old and E_new (E_old - E_new specifically) is, the smaller the probability will be.

We use the below configurations to plot the graph partitioning result:

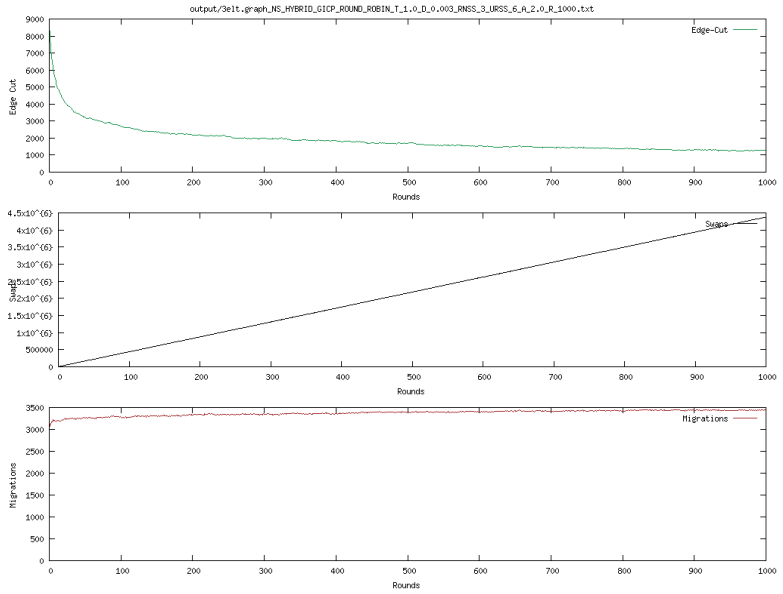
- Dataset = 3elt
- K = 4, K-way graph partitioning
- Sample polity = H, hybrid heuristic
- SA method = [previous exponential SA, newly defined exponential SA]
- T0 = 1
- delta = 0.003

Here is the result plot:

- Plot with precious exponential simulated annealing function:



- Plot with newly defined exponential simulated annealing function:



In conclusion, our new function will provide a better edge-cut result while requiring much more time to reach a convergence in number of swaps.